Designing parameterizable hardware IPs in a model-based design environment for high-level synthesis

Designing Parameterizable Hardware IPs in a Model-based Design Environment for High Level Synthesis

Shahzad Ahmad Butt, National University of Computer and Emerging Sciences, shahzad.butt@nu.edu.pk
Mehdi Roozmeh, Politecnico di Torino, mehdi.roozmeh@polito.it
Luciano Lavagno, Politecnico di Torino, luciano.lavagno@polito.it

Model-based hardware design allows one to map a single model to multiple hardware and/or software architectures, essentially eliminating one of the major limitations of manual coding in C or RTL. Model-based design for hardware implementation has traditionally offered a limited set of micro-architectures, which are typically suitable only for some application scenarios. In this paper we illustrate how DSP algorithms can be modeled as flexible intellectual property blocks, to be used within the popular Simulink model-based design environment. These blocks are written in C, and are designed for both functional simulation and hardware implementation, including architectural design space exploration and hardware implementation through high level synthesis. A key advantage of our modeling approach is that the very same bit-accurate model is used for simulation and high-level synthesis. To prove the feasibility of our proposed approach, we modeled an FFT algorithm and synthesized it for different DSP applications with very different performance and cost requirements. We also implemented a high level synthesis IP generator that can generate flexible FFT HLS-IP blocks that can be mapped to multiple micro/macro-architectures, to enable design space exploration as well as being used for functional simulation in the Simulink environment.

Categories and Subject Descriptors: B.5.2 [**REGISTER-TRANSFER-LEVEL IMPLEMENTATION**]: Design Aids; B.5.2 [**SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS**]: Real-time and embedded systems

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: model-based design, model-based high level synthesis, simulink modeling, parameterized IPs, IP generator, design reuse, audio detector, GPS acquisition, C/C++ hardware IP description, FFT

ACM Reference Format:
Butt, S.A., Lavagno, L., Roozmeh, M. 2015. Designing Parameterized Signal Processing IPs for High Level Synthesis in a Model Based Design Environment. ACM Trans. Embedd. Comput. Syst. 10, 1, Article 22 (August 2015), 28 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Model-based design environments (MBDEs), such as Simulink, are becoming more widespread as they expand their capabilities of synthesizing efficient hardware and software from high-level algorithmic models. They find application in very important areas such as digital signal processing (DSP), telecommunications, and control systems. MBDEs allow modeling of complex algorithms and systems at a very abstract level, using pre-defined primitive micro and macro blocks (e.g. adders, multipliers, multiplexers, FIR filters, FFTs). The designer can thus focus on defining the best algorithm without caring for tedious low level implementation details. Such details can be introduced later in the design flow via automated model-to-model translation, including both direct mapping and sophisticated hardware and software synthesis algorithms, or through a user-directed refinement process.

One of the major advantages of MBD tools is that they let the designer verify and validate abstract golden models against their design specifications. The designer can then use these models to generate code targeting either a specific embedded processor for software implementation, or a register transfer level (RTL) description for hardware synthesis. The process of generating software code or the RTL hardware description is

normally assisted by the designer by providing constraints and directives. Algorithmic design provides a much better scope for power, area and performance optimizations as compared to what can be achieved at lower levels. MBD also greatly eases the verification task by allowing one to re-use already verified macro blocks and more importantly by letting the designer use the same verified golden reference model throughout the complete design, verification and implementation flow.

State of the art MBD tools used in the industry can generate very efficient and optimized software code for different target processors, by using information about the target processor architecture. But hardware implementation essentially entails the generation of a cycle accurate RTL model from very abstract block level models that have no notion of clock cycles. Hence the set of choices is much broader, and the normal direct translation strategy used for software implementation is likely to fail. Current MBD tools, such as Simulink from The Mathworks, can generate a very limited set of hardware implementations starting from a given model. In other words, they have limited capabilities to explore the hardware design space starting from a single model, due to reasons that are described more in detail in the next sections. This is particularly true in the case of complex blocks like a Fast Fourier Transform (FFT), a Discrete Cosine Transform (DCT) or a Viterbi decoder, which are normally represented as Simulink macro blocks.

Simulink is one of the best known and most extensively used MBD tools. It has a rich library of components that can be used to model systems and algorithms from many different domains. In Simulink libraries, the components are arranged in groups known as blocksets, for example the DSP-blockset that can be used to model DSP algorithms. Simulink libraries are extensible through a mechanism known as S-functions. It provides a component modeling paradigm in which the functionality (algorithm) as well as the interaction with other components can be represented in a well-defined way. The S-functions can be written in C, FORTRAN, or MATLAB, as required.

Simulink comes integrated with a tool called Real Time Workshop (RTW). RTW is a set of code generators known as target language compilers (TLC) that can translate a Simulink model to C/C++. Each TLC can be optimized to generate code for a different processor or platform. Embedded Real Time (ERT) coder is one of these TLCs, which is optimized to generate software code for embedded applications. It can generate floating and fixed point code. Simulink models can be also translated to RTL description for hardware synthesis through a tool called HDL Coder. Efficient hardware implementation starting from an abstract model generally requires effective design space exploration (DSE) from a single model. HDL Coder, however, has limited capabilities in this regard, especially when it comes to complex algorithms like FFT, DCT, and Viterbi decoders. Each HDL coder block is mapped to a few micro-architectures, e.g. fully sequential and fully pipelined, which provide only a few design points, such as minimum area or maximum throughput. Many of the architectural trade-offs that are essential for optimized hardware implementation, such as independent definition or throughput and latency, or the choice of memory parallelism and architecture, may even need to be performed manually, by changing the source model every time. This changing of model defies one of the main purposes of model-based design, by requiring different models for different implementations, and hence making the design process long and tedious.

It was shown in recent work [Butt and Lavagno 2012a][Sayyah et al. 2011] that effective hardware synthesis and design space exploration can be performed starting from Simulink models. Simulink can be used as a modeling front-end to high level hardware synthesis tools, which take as input functional specifications in the form of C/C++ or SystemC. The strategy that was used consisted of first translating a Simulink model to C/C++ using RTW, and then using this as input to high level

hardware synthesis after proper definition of hardware interfaces. The connection between the Simulink model and high level synthesis (HLS) was thus obtained in that work by using the automatic C code generation capability provided by RTW. The added advantage that comes through the use of a high level hardware synthesis tool is that hardware design space exploration can be easily performed starting from same single model, with little modification of the functional C code generated from the Simulink model.

From our experience, however, we observed that the C code generated by RTW can be used efficiently for hardware synthesis only if it involves mostly simple blocks. This is the case when the algorithmic design is performed using fine-grained blocks. But when it comes to using complex macro block like Fast Fourier Transform (FFT) or Discrete Cosine Transform (DCT), then the software-oriented C code that is generated by RTW limits the hardware design space that can be explored. This is because the structure of the SW-oriented C code used to model such blocks can be based on a signal flow representation that inherently limits, as we will argue below, the kind of micro-architectures that can be explored. In this paper we propose the modeling of such complex macro-blocks still using plain C, which is essential for smooth integration with Simulink-based verification, but using a code structure that lends itself to better HW design space exploration as a parameterized high-level Intellectual Property (IP) block. Figure-1 illustrates in detail the integration of this kind of hardware-oriented IP in the Simulink model-based design flow for high level hardware synthesis.

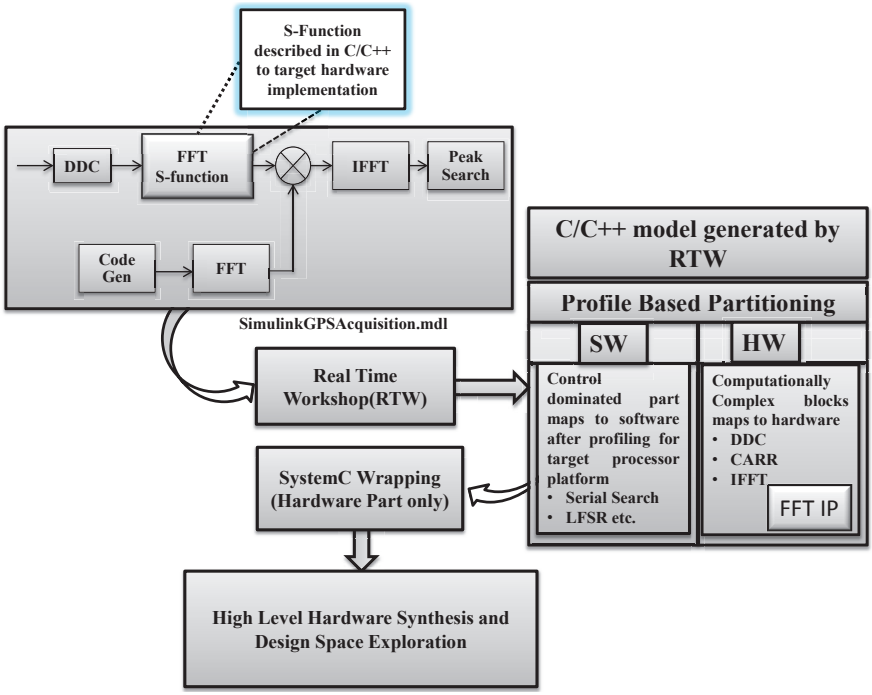

Fig. 1. Complete Model-based design flow using the FFT S-function IP

We need a specialized C model because state of the art HLS tools can effectively explore the micro/macro architectural design space starting from a single model only if it explicitly exposes parallelism and memory architecture in a form that can be used

(e.g. by unrolling, pipelining and merging loops) to efficiently map code blocks (functions/loops) to concurrent hardware units. ERT deals with complex blocks like FFT, DCT, and Viterbi decoder as atomic blocks and does not have a suitable representation with enough parallelism to be used for HW design space exploration.

The key idea of our approach is to use a single functional model, composed of several C functions, which can be scheduled both as parallel threads and as a single thread, and which can use a variety of memory architectures (register file, single-ported or dual-ported memory), thus providing a very broad area/performance/power trade-off space.

2. RELATED WORK

Simulink models have been used for hardware synthesis and hardware/software co-design in the past. For example, [Toledo et al. 2006] presents a Simulink image processing blockset for hardware/software co-design, using Xilinx System Generator as the backend for hardware generation. The authors have developed a component library that can be used for modeling image processing algorithms. The Xilinx System Generator has been used to translate Simulink models to RTL. It has limitations that are more or less similar to those of HDL Coder. In [Haubelt et al. 2008] an ESL design tool is presented that can do automatic design space exploration starting from behavioral SystemC models, but the modeling is completely done in SystemC, whereas we propose to use Simulink as a modeling environment . [Huang et al. 2007] presents a design flow for mapping a Simulink model to a full Multi-Processor System-on-Chip. The design flow allows processor and task design space exploration at various abstraction levels, but does not provide any support for mapping part of the model to dedicated hardware.

In [Butt et al. 2011] we discussed how to configure RTW for generating C code from Simulink models and also to how to wrap the automatically generated code into a SystemC wrapper that can be used for hardware implementation. We focused on how to tune RTW for generating code that is suitable for HLS and on how to obtain different points in the design space without requiring a deep understanding of the automatically generated SystemC code. On the other hand, this new paper discusses how to model complex algorithms in a way that allows greater reuse of the same model in different application scenarios, by enabling design space exploration and hence multiple implementations starting from the same single model.

In [Wernsing and Stitt 2010] the authors define a modeling style that allows for efficient HW/SW tradeoffs. On the other hand, we use Simulink, that is a well-known industrial implementation of a similar data-flow modeling style. [Kienhuis et al. 2000] presents a tool named Compaan that automatically transforms a nested loop program written as MATLAB Script or C code into a process network specification. It can extract parallelism from MATLAB and C code, but requires code rewriting to fit the Compaan modeling style (affine array indices within nested loops without control), while in our case we exploit designer-provided top-level parallelism among Simulink blocks.

Simulink HDL Coder generates synthesizable RTL code from Simulink models. However, the generated RTL code has a close 1-to-1 correspondence with the Simulink model, hence different micro-architectures in hardware require very different Simulink models, thus defeating the separation between functionality and architecture that is essential for true model-based design. Moreover, most architectural trade-offs like resource sharing, pipelining and exposing more parallelism are tedious to model in Simulink. Recent versions of HDL coder include options to choose the micro-architecture of complex blocks like FFT or DCT among a very small set of options. For FFT this includes only two extreme options, namely fully resource shared with low throughput structurally pipelined with streaming interfaces (one sample per

cycle). This is clearly a move in the right direction, but we argue, and will show experimentally in the last section, that our strategy generates competitive results, and is based on a much more flexible design flow, including easier interfacing with other HW blocks, and exploration of a much broader design space from a single model.

Another approach that is normally used by FPGA vendors is to provide the end users with an IP generator that can take as input different parameters and then produce an RTL description that is tuned for their FPGA architectures. For example in [Mitra 1999] the authors discuss the design of a tool for creating such IP cores in VHDL. The tool makes it possible for third-party IP developers to create cores targeted to XILINX FPGAs. For FFT Altera has a tool called FFT MegaCore Function and Xilinx has an FFT block called FFT LogicCore that can generate RTL for different FFT IPs, but they mostly rely on either a fully concurrent streaming architecture or a very sequential resource shared architectures.

Our goal, on the other hand, is to represent IPs in a flexible way, at a high level of abstraction, to allow mapping with different architectures and levels of concurrency. We also ease the task of system verification by simulation, by making our models integratable into Simulink as S-functions. Once the model is integrated as an S-function into the full Simulink system model, the entire system can be verified at the algorithmic level and can generate a set of golden vectors for RTL verification.

Since this paper uses the FFT as a commonly used IP block that needs to be modeled specifically for HW implementation, we now review papers devoted to parameterized HW FFT models. Traditionally this kind of IP blocks are designed [Lee and Chen 2006a][Chouliaras et al. 2009] at the Register Transfer Level, in order to be directly used for low level implementation steps like logic synthesis and physical design. Many architectures exist that can be used for optimized implementation in different application scenarios. They define the level of concurrency and the computation parallelism for various scenarios and also provide techniques to optimize the required memory architecture. However, these IPs offer very little room for architectural optimizations/changes required by specific application scenarios. Only in a very few cases, such as those described in [Murphy et al. 2004], the data-path and the algorithmic configuration can be parameterized.

Little work has been performed in the domain of high level synthesis-IPs (HLS-IPs). In [Takach 2010] the authors discuss the design of parameterizable FFT IPs. They focus on representing the design using advanced C++ templates and tool-specific data type libraries to describe bit accurate hardware. However, Simulink cannot use C++ templates and would have difficulties incorporating external libraries within S functions. On the other hand, *our parameterized IP blocks are modeled in plain C, to be compatible with the Simulink S-function modeling style*. The authors in [Dave et al. 2006] also considered architectural exploration using BlueSpec SystemVerilog, but they did not fully exploit the power of high-level synthesis, since the micro-architecture is flexibly coded in their SystemVerilog model. Moreover, SystemVerilog cannot be used inside Simulink for algorithmic verification in the same way as our C model can.

In [Kee et al. 2008] the authors discuss an approach that is related to ours in terms of functional modeling. They present a technique that allows one to generate different FFT IPs with different levels of concurrency and parallelism. The generated FFT IPs are LabView [Kee et al. 2008] data-path diagrams that can be synthesized to RTL using an FPGA synthesis tool that is a part of the LabView design suite.

3. PROPOSED DESIGN FLOW FOR HLS-IPS

Figure-2 shows our proposed design flow for HLS-IP integration, verification and high level synthesis. In our recent publication [Butt and Lavagno 2012b] we proposed a design flow that allows one to model hardware for complex algorithms in plain C for

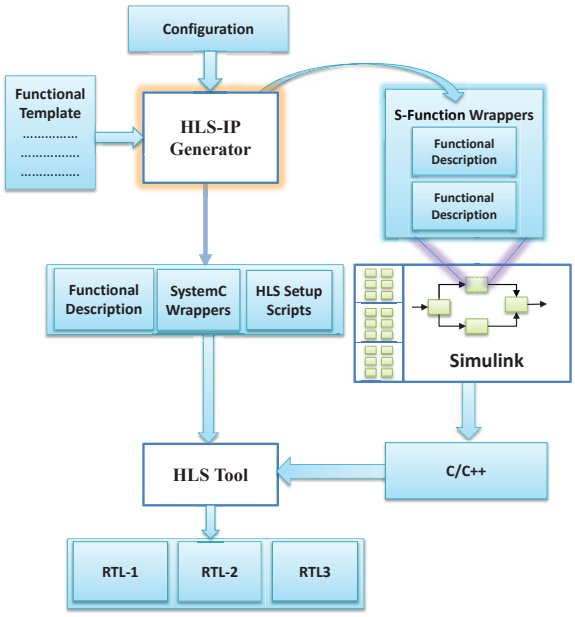

Fig. 2. FFT HLS-IP generation, verification and synthesis design flow

integration in the Simulink model-based design environment. We also discussed how to use the same functional C code for efficient synthesis with a high level hardware synthesis tool, which accepts algorithmic specification in C/C++ as input. FFT block was used as case study. In that paper we used the FFT as a case study to highlight the importance of algorithm partitioning and selection of concurrency during high level synthesis to obtain optimized implementations. In that work the focus was mostly on figuring out how to define algorithms like the FFT in plain C for integration in the Simulink environment, and how to differentiate and choose between interesting signal flow graphs for hardware implementation that offer better opportunities for design space exploration. The FFT algorithm was modeled to accept variations in terms of data-path size, transform length (number of samples) and, up to some extent, micro-architecture. The way in which the FFT was modeled only allowed one to synthesize hardware that used a shared memory and had a fixed concurrency.

In this paper, on the other hand, we discuss a modeling style for HLS-IP (which is again an FFT, due to its widespread use) that allows one to synthesize hardware implementations that can have different level of concurrency and micro-architecture. Our key idea is to generate a partitioned model of the FFT, where different partitions can be merged in order to obtain different implementations with different cost and performance as required. In this new paper we also present an HLS-IP generator that automates many design steps that were left manual in that earlier paper.

The complete design flow is shown in Figure-2. The figure shows that the HLS-IP can be generated automatically based on a configuration file. Note, however, that the same approach can be used for HLS-IP integration, simulation and synthesis of a manual description written following the guidelines and constraints given in this paper. The first step that must be completed by a user of our IP is to write a small configuration file that defines various parameters, such as the length of the FFT, the widths of the data-path layers, and the type of SystemC wrappers to be generated (if required by the HLS tool). The FFT HLS-IP generator produces:

(1) the algorithmic description in C,
(2) the S-function wrappers that allow the integration of the generated C-code with Simulink models for simulation and verification,
(3) the SystemC wrapper (if required), and
(4) a set of HLS scripts that can map the generated IP to various micro/macro architectures to ease the hardware design space exploration.

Once the set of scripts and the C model are generated, the next steps consist of:

— Using the S-function wrapper and the generated code for simulation-based verification in Simulink. This step uses the full power of the Simulink/Matlab modeling environments to find the best overall algorithmic solution to the problem at hand. In this phase also the arithmetic precision that is required to satisfy the application requirements is identified. We provide several examples of such algorithmic models in Section-7.
— Performing HLS and design space exploration starting from the HLS-IP functional code. The FFT HLS-IP will in most cases appear as a separate hardware block in the final implementation, unless the performance requirements are really low, and it can be scheduled with other functionalities in order to save area. If the other parts of the model also need to be implemented in hardware, then they can also be mapped to hardware by generating C-code from the Simulink model and then using this code as input to the C-to-RTL HLS tools, as discussed in detail in [Sayyah et al. 2011].

4. HLS-IP MODELING STRATEGY

Modeling HLS-IP as plain C code that still enables HW design space exploration through HLS can be challenging. Even though HLS can vary several micro-architectural parameters, such as architectural parallelism, loop pipelining, resource sharing, memory splitting and merging, and so on, several of these options are available only when the appropriate modeling style is used in C. For example in DSP applications the data-path width is decided based on the results of several bit-accurate simulations (e.g. using the fixed point optimization capabilities of Simulink). However, representing bit-accurate types in plain C can be tricky. Moreover, DSP applications derive most of their performance from both architectural level and fine-grained parallelism and pipelining.

We developed a modeling strategy based on a representation which can be easily used to explore different micro-architectures. It also accurately models different data-path bit widths and arithmetic overflow/saturation modes in a single C model, which is both compatible with the S-function modeling style and amenable to efficient HW synthesis. This strategy is illustrated in this paper, for the sake of explanation, with an FFT algorithm, but it can be easily applied to a more general class of DSP algorithms (e.g. DCT, Viterbi, etc.). Also blocks that access memory in a non-linear fashion, such as cache controllers, and non-linear kernels [Butt et al. 2014] would benefit from such a synthesis-oriented Simulink HLS-IP approach. During partitioning the general notion of maximizing throughput and simplifying interface cost should be followed. But one should always keep in mind that the partitions and interfaces should be defined in such a way that they can be merged depending on the target application throughput requirements, which may result in resource savings.

4.1. Signal Flow Graph Representation and Algorithm Partitioning

Modern state of the art HLS tools are very good at scheduling operations onto a shared resource architecture, but they have only limited capabilities to extract and expose parallelism in sequential descriptions. So a very important first step during the design

of HLS-IPs is to select a suitable signal flow graph (SFG, also known as data Flow Graph) that allows one to expose parallelism in an explicit way.

The Fourier Transform is widely used in signal processing to transform signals from the time domain to the frequency domain and vice-versa. The Fourier Transform that operates on discrete data is called Discrete Fourier Transform (DFT). The Fast Fourier Transform (FFT) is one of the most famous and widely used algorithms to calculate the DFT and its inverse. The FFT algorithm exploits the symmetry of the calculation and the re-use of already performed calculations to reduce the computation complexity from N^2 to $NLog_2N$ for a DFT that is computed on N samples. The FFT algorithm is selected as case study because it can be represented using different non-trivial SFGs and finds application in many signal processing areas. These various signal flow graph representations are beneficial for targeting different application domains with different performance requirements under different constraints. Our target is to derive all those that required to cover a very broad design space from a single functional model in C.

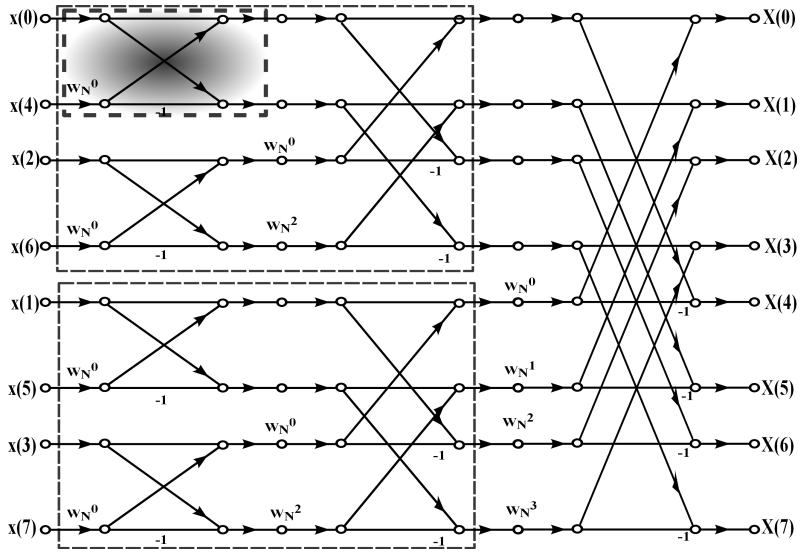

Fig. 3. Signal flow graph for radix-2, 8-point in place FFT computations

Figure-3 shows a signal flow graph (SFG) for computing an FFT with 8 samples. The SFG represents the fully unrolled computations and data dependencies (and thus the full available parallelism) implied by the C code structure used by RTW for a software-oriented FFT implementation. In this SFG each node represents a complex operator and each arc represents a complex value. It is called radix-2 FFT since its basic unit, called butterfly and marked by the dotted box at the top left of the figure, consumes two input samples to produce two output samples. Constants marked as W_N^0, W_N^1, W_N^2 and W_N^3 are complex exponentials, known as twiddle factors. Inputs $x(0)$, $x(1)$, $\ldots x(7)$ are the complex time domain samples of the signal to be transformed and outputs $X(0)$, $X(1)$, $\ldots X(7)$ are the complex values of the frequency spectrum of the signal. Each butterfly represents the multiplication of twiddle factors by input samples and then one addition and subtraction to calculate outputs.

The signal flow graph in Figure-3 is called in-place FFT because every butterfly can write outputs to the same memory from where it has read the inputs. Such a represen-

tation is useful for implementing a resource shared FFT with relatively low throughput requirements, targeting low power applications with limited on chip memory size and bandwidth. But this kind of signal flow graph is not well suited when throughput requirements are high and either a pipelined implementation or a fully unrolled register-based (rather than memory-based) implementation is required. For example, let us assume that in order to increase throughput we unroll the inner loop that performs butterflies in a stage (a column of Figure-3), and that stage inputs are mapped to registers. After performing a butterfly computation, the inputs for the next butterflies mapped to the same multiplier/adder/subtractor resources will come from signal flow graph positions that are different from the first stage, which in hardware will imply high multiplexing cost and hence will not be efficient. Similarly, some tools and memory architectures may not efficiently support pipelining of loops in which computations read and write from the same memory, due to the need to use multi-ported memories. This, on the other hand, would be easy in software, for which the graph in Figure-3 works best.

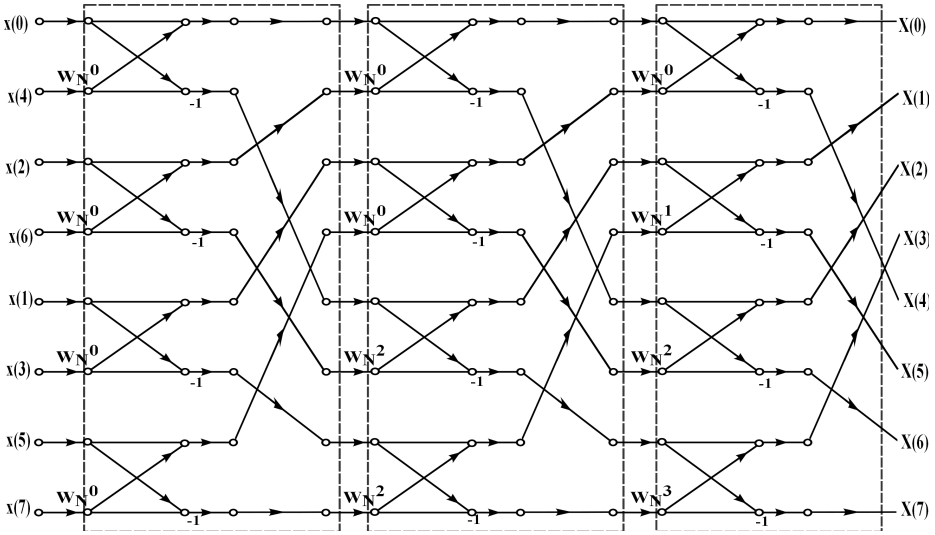

Fig. 4. The Signal flow graph for radix-2, 8-point FFT computations

Figure-4 shows another signal flow graph for FFT, which also can be represented in C in the form of nested loops, but is much more flexible than the one in Figure-3 to derive many possible implementations using HLS. In particular, it can be mapped to a register-based unrolled implementation.

The advantage of such an FFT representation with respect to Figure-3 is that the interconnection network between the stages is the same for all the stages, which results in less multiplexing cost when a stage is partially or fully unrolled and subsequent stages are implemented by iteration. Even when a memory-based implementation with more aggressive resource sharing and lower throughput is required, still the signal flow graph in Figure-4 is more flexible. This is because one can always map inputs and outputs of a butterfly to two different memories, while still allowing partial unrolling depending on the memory read/write bandwidth. The signal flow graph in Figure-4 can even be mapped to a single on-chip memory implementation by utilizing the memory merging capabilities offered by HLS tools, which allows one to map two different memories of different lengths and widths (arrays in C) to a single aggregate

memory. In our FFT HLS-IP we used the SFG in Figure-4, because it can offer broader design space exploration as compared to Figure-3, which corresponds to the default software implementation from Simulink RTW.

4.2. Parameterizable Datapath Modeling

The datapath of an HLS-IP should be modeled using parameterizable arithmetic data types. Which allows to appropriately select bit-widths and different point in datapath. Another constraint on these data types is that they should be described in plain C (not C++) to make them usable in Simulink C S-functions. The following discussion uses FFT butterfly units as example. A detailed representation of the signal flow graph of a radix-2 butterfly is shown in Figure-5, where values and operators are on real, rather than complex, numbers. Here X0r, X0i and X1r, X1i represent inputs, Y0r, Y0i and Y1r, Y1i represent outputs and WNr, WNi represent twiddle factors. Superscripts 'i' and 'r' identify the real and imaginary parts respectively. For hardware implementation this local signal flow graph must be represented as a fixed point data-path with specific bit widths. Based on our design flow requirements, the same representation must be usable for HLS and for simulation in Simulink. In the next section we discuss how we modeled fixed point operators with different arithmetic modes to satisfy all these requirements.

4.2.1. Modeling Arithmetic Operators for HLS. Fixed point operators take two inputs and produce an output, each with a given bit width, location of the decimal point, and rounding and overflow mode. Some pre and post processing steps, such as decimal point alignment, rounding and overflow management are necessary to correctly perform arithmetic operations. We divided our C-based implementation of each fixed point arithmetic operator in three steps.

— core operation (including alignment),
— rounding,
— overflow management.

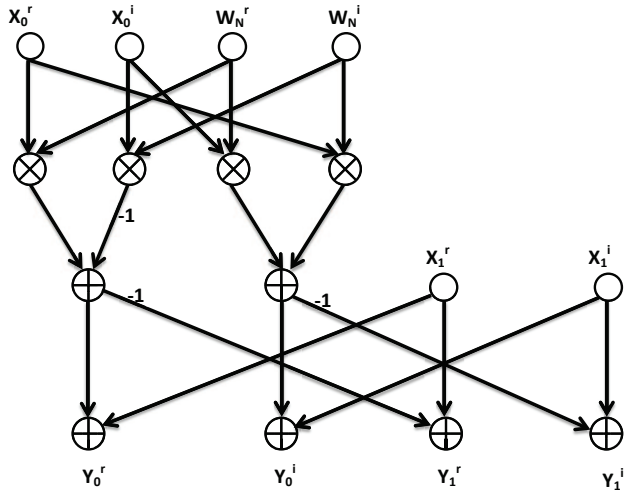

Fig. 5. Figure 4. Radix-2 butterfly signal flow graph

Alignment is included in the core operation because it depends on the operation (e.g. for addition or subtractions inputs need to be aligned, while for multiplication they do not).

Rounding, overflow management and core operation are modeled as reusable plain C functions. For example, the overflow manager function (implementing wrapping and saturation) is called when any addition, multiplication or subtraction operation requires an output to be stored in fewer bits than the full bit width of the core operation result, which can be computed from the input operands. The rounding function on the other hand (implementing truncation, ceiling, floor and rounding) is called when the number of bits for the output fractional part is reduced. Finally, functions that are used to model the core operation take as input the operands with bit width and decimal point information, and they produce an output with the desired bit width and decimal point. They call rounding and overflow functions as needed.

All these functions are automatically inlined during hardware synthesis, and since the precision, point position, rounding and overflow selection arguments are synthesis-time constants, the high-level synthesis tool can perform efficient bit width inference for all needed hardware resources.

Note that this method is only applicable if the total bit width for each input or output (including temporary outputs before overflow and rounding management) is less than or equal to the maximum integer size supported by the machine where Simulink and the high-level synthesis tool are executed (typically 64 bits).

Normally HLS tools use data types, like sc_fixed or ac_fixed, which use C++ templates to make arithmetic expression representation and automated conversions and casting easier and more natural to handle. However, such sophisticated mechanisms are not available in the plain C which is required by our methodology, since we do not want to use different models for verification and hardware synthesis, which would require one to re-verify the HW-oriented models to be used by the high-level synthesis tool. However, we observed that if masking coupled with sign extension is applied at appropriate places in the data-path, then the HLS tool can identify and optimize the data widths of the allocated resources even with our plain C representation of fixed-point data types. Moreover, *we are advocating this style only for frequently re-used IP blocks, where hardware implementation flexibility and efficiency are more important than ease of modeling within the blocks*. Figure-6 represents the flow chart of a fixed point addition operation.

Note that for an FFT, butterflies in the same stage have the same data-path and produce outputs with the same fixed point representation. But butterflies in different stages can have different bit widths, which can be handled either by increasing the width by one at each stage, or by appropriate rounding. We used the latter technique, since it lends itself to better resource sharing among stages, and is commonly used in practice. The fixed point parameters passed to the various arithmetic operators can thus be derived from input bit width, output bit width and length of the FFT.

4.3. Simulink S-functions

Simulink allows one to extends its components libraries through a powerful mechanism called S-function. S-functions are computer programs defined in different languages like C, FORTRAN and M-script. These S-functions are written conforming to a set of requirements provided in the Mathworks documentation. In brief, each Simulink block to be modeled as an S-function must implement or use a list of functions (API) that allow it to interact with the simulation kernel. Depending on the type of block (continuous or discrete time, with state or without), these are called by the kernel to initialize the block, execute one simulation step, and terminate the block. The block, on the other hand, can call functions to access its inputs, outputs and state.

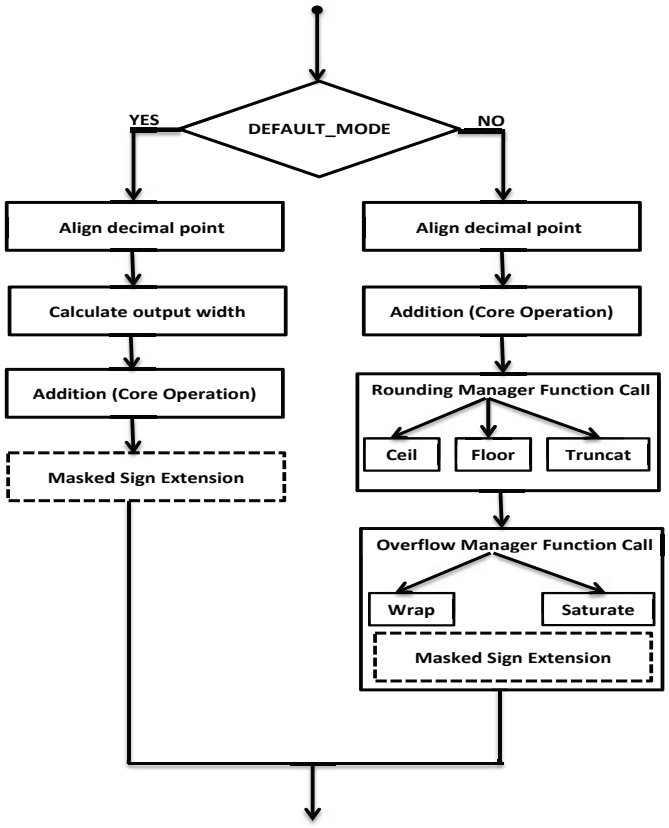

Fig. 6. Flow chart for addition operation with masked sign extension

Figure-7 illustrates how the Simulink engine interacts with the blocks, both S-function based or native. Initialization is carried out at the start of the simulation by calling a function provided by each block to initialize state or global variables. Then simulation enters a loop in which it calls two other block functions to calculate outputs and update states. Simulink provides a unified access to inputs, outputs and other parameters of an S-function block through a structure called SimStruct. The members of the structure can be accessed for reading and writing through a set of functions provided by Simulink.

The code listing in Algorithm-1 shows a simplified S-function wrapper. Here mdlInitializeSizes is a function which is called by the simulation engine for initialization and mdlTerminate is called when the simulation ends. The function mdlOutputs is used to update the output of any block during simulation. There is also a function called mdlUpdate which is used to update state which is carried across multiple block output update iterations. For our FFT HLS-IP there is no state to be carried across multiple iteration because it calculates the transform of a complete frame in single invocation. Simulink also provides a graphical user-interface called S-function builder. It takes as input the user-defined C/C++ files and some other information related to block states, inputs, outputs and it generates this wrapper automatically.

ALGORITHM 1: Simplified S-function wrapper

```
#define S_FUNCTION_NAME  USER_DEFINED_SFUCNTION_NAME
#define S_FUNCTION_LEVEL 2
#include "simstruc.h"
static void mdlInitializeSizes(SimStruct *S)
{
}
static void mdlTerminate(SimStruct *S)
{
}
static void mdlOutputs(SimStruct *S)
{
  ...
        //Read input from SimStruct
        functional_behavior_stage1(....);
        functional_behavior_stage2(....);
        functional_behavior_stageN(....);
        // Write Output to SimStruct  ...
}
<additional S-function routines/code>
#ifdef MATLAB_MEX_FILE #include "simulink.c"
#else #include "cg_sfun.h"
#endif
```

5. MODELING FFT AS HLS-IP

Most model-based design environments offer the FFT as a built-in macro block. They can also generate C code for software and RTL for hardware implementation. The C code, as mentioned above, is optimized by exploiting some knowledge of the processor architecture. But when it comes to hardware, the implementation requirements can be very diverse, ranging from a low throughput low power sequential architecture to a high throughput highly concurrent architecture. The FFT macro block that is normally offered by a model-based design environment relies on a pre-defined fixed architectural template that can be parameterized to change the data-path width and the FFT length. As argued above, this may result in a sub-optimal hardware implementation depending on the specific architecture support and the application area to be targeted. In this section we discuss how we represented the FFT as a parameterized C IP block that can be used for verification in Simulink and for HLS using a C or SystemC-based tool.

5.1. C modeling of the FFT HLS-IP as S-function

Simulink integration of the FFT HLS-IP is performed by implementing and using several C functions defined by the S-function APIs described above.

Our FFT model is written in plain C and split in different files, as shown in Figure-8, in order to make it more understandable and ready for easy encapsulation in an S-function wrapper or SystemC wrapper.

— The “params.h” file defines all the constants to model the fixed point data-path, the parameters of the FFT computation, such as for example the FFT length, radix, as well as the bit width and point position for inputs and outputs. It also allows one to switch between fixed point and floating point in order to ease fixed point conversion.
— The “globals.h” file defines all the global variables, such as the buffer memories, which should be defined as public members when encapsulated in SystemC.

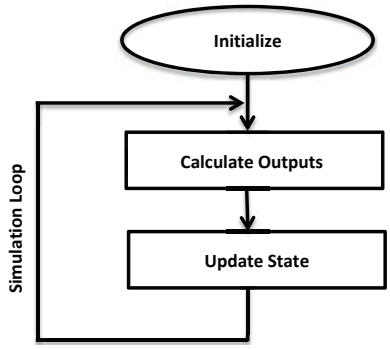

Fig. 7. Interaction of Simulink simulation engine with S-function blocks

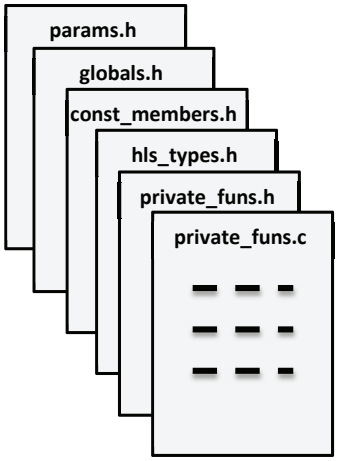

Fig. 8. IP C code organization in different files

— The "const_members.h" file defines all constant data, such as the twiddle table defining all the complex twiddle factors that are required to perform butterflies in different stages. Unfortunately they are different for every FFT length. Then the twiddle table must be (automatically) pre-computed and saved as a C text file used to initialize a constant array for any given parameterization of the IP.
— The "hls_types.h" file conditionally defines basic data types as floating point or integer, as selected in "params.h".
— The "private_funs.c" and "private_funs.h" files define and declare all the other functions required to implement the FFT, including the top level core wrapper function that iterates over butterflies to implement the FFT execution.

5.2. SystemC wrapper for the FFT HLS-IP

When I/O signal names and bit widths and the communication protocol are selected for the FFT HLS-IP, then a SystemC wrapper can be generated for HLS. This wrapper serves two purposes; it defines the synthesizable RTL interface of the block and it integrates the plain C code in a class structure (an `SC_MODULE`), where global variables appear as public members and all constants appear as constant public static members. It also defines the SystemC thread behavior by appropriately calling the

```
#include "params.h"
#include "hls_types.h"
#include "private_funs.h"

class FFT: public sc_module {

public:
 #include "globals.h"

 sc_in<bool> clk;
 sc_in< bool > reset;
 sc_in< bool > start;
 sc_in< sc_int <INPUT_WIDTH> > data_in_real;
 sc_in< sc_int <INPUT_WIDTH> > data_in_imag;
 sc_out< sc_int <OUTPUT_WIDTH> > data_out_real;
 sc_out< sc_int <OUTPUT_WIDTH> > data_out_real;

          void core_wrapper_function (------);
 }
```

Fig. 9. SystemC wrapper for IP

top-level function defined in "private_funs.c", as shown in Figure-9. The S-function I/O interfaces are implemented as two separate cycle accurate functions (also known as "transactors") that inherit the I/O data types from "hls_types.h". For the FFT HLS-IP we used a simple streaming I/O protocol with handshake signal to initiate data transfers. Note that generation of this boilerplate code can be easily automated with a script. This SystemC wrapper is not required by some high level synthesis tools, for example Catapult from Calypto. In our case we first started experimenting with CtoS from Cadence, so we also implemented a SystemC wrapper generator.

5.3. S-function Wrapper for the FFT HLS-IP

The S-function wrapper can be generated in two ways. First it can be written manually, by following all the rules described by the MathWorks. As an alternative, Simulink provides a graphical environment, called S-function builder, that can be configured to generate the S-function wrapper automatically. In this case only a few lines of manually written code are required to describe how the top level function that implements the FFT kernel accesses the input and output (the FFT block has no state to be carried across executions).

5.4. Basic Template Hardware Architectures for FFT

Two kinds of basic architecture templates are utilized in most cases [Lee and Chen 2006b][Nikolic 2011]. One is a resource shared architecture template shown in Figure-10, where the LUT block identifies the twiddle factor generator. In this architectural template, the complete FFT is mapped to a set of butterfly computation units. These units normally fetch data from a large memory that stores all the input samples and all the intermediate results. When two memories are used, they are required to be swapped as input and output after the execution of butterflies in a stage. The number of butterfly units that can operate in parallel to execute butterflies in a stage depends mainly on the bandwidth of these memories, which in many cases is limited. This

architectural template can be used for applications with low throughput requirements and for implementations that use less chip area.

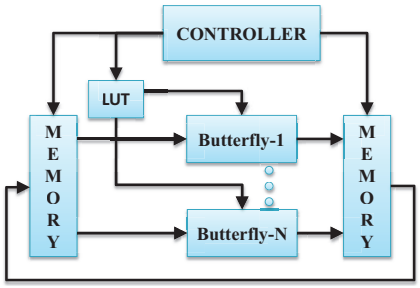

Fig. 10. Resource shared hardware architecture for FFT

The other architectural template that is mostly used for high throughput applications, is called pipelined FFT architecture. It has many variants but in its fundamental form it looks like the one shown in Figure-11. It uses a butterfly unit and a twiddle factor generator per butterfly stage. For example, the signal flow graph shown in Figure-12 for an 8-sample FFT uses three butterfly stages. This usually consumes more hardware than the resource shared architecture.

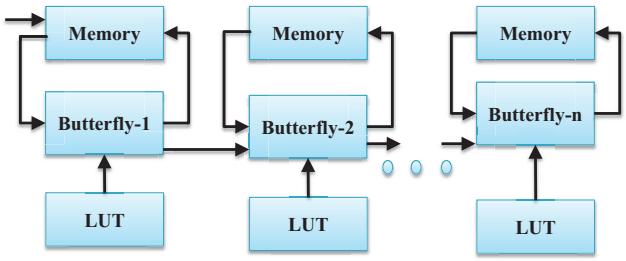

Fig. 11. Pipelined hardware architecture for FFT

The resource shared architecture doubly folds the FFT signal flow graph both horizontally and vertically. The pipelined architecture on the other hand folds it only in the vertical directions, to map each butterfly stage to a concurrent hardware unit. In [Kee et al. 2008], the authors present an interesting approach that implements a hybrid of vertical and horizontal folding to generate different kinds of interesting architectures, but its detailed discussion is not in the scope of this paper.

6. FLEXIBLE HIGH LEVEL DESCRIPTION OF HLS-IP

In our past work [Butt and Lavagno 2012b], the FFT was used as a case study for application domains in which a doubly-folded resource shared architectures was sufficient for implementation. The core of the FFT kernel was specified as a double nested loop in C which resulted in different single threaded architectures (aggressively resource shared) with variations in terms of micro-architecture, memory bandwidth and data-path width.

However, when high levels of concurrency are required to meet application requirements, the architecture of the memory also becomes important, as argued above. For concurrent implementation the memory is usually a set of distributed buffers that can

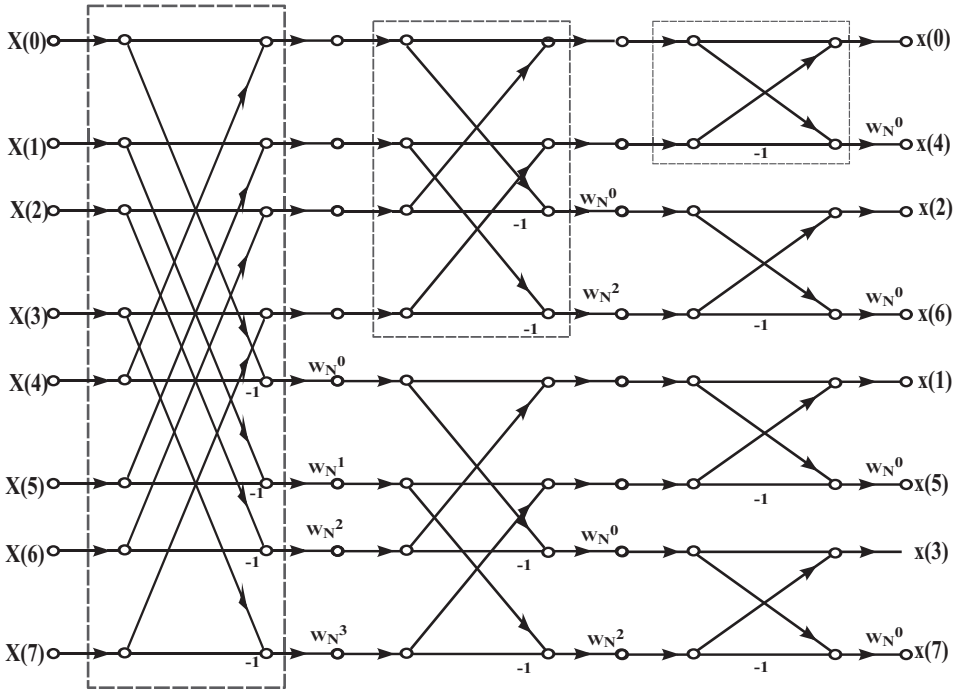

Fig. 12. . Radix-2 FFT signal flow graph

be mapped to SRAMs during HLS. This is because every concurrently executing hardware block requires a dedicated access to memory in order to satisfy the read/write bandwidth requirements. Hence *in this paper we propose a new IP modeling strategy that allows a variable level of parallelism to be extracted from a single plain C functional model.*

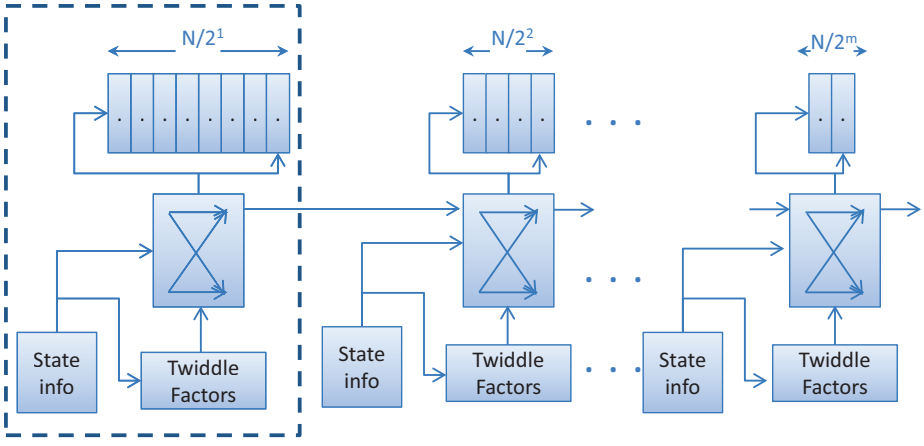

Fig. 13. Symmetry of Pipelined FFT Architecture

The pipelined FFT architecture shown in Figure-13 is very symmetrical. The dotted box represents the folding of a single butterfly stage of the FFT signal flow graph. Folding essentially maps a stage to:

— a buffer that acts as a shift register,
— a butterfly computation unit (the core data-path),
— control information, which essentially describes which butterfly in this stage is executing at any time,
— a set of twiddle factors.

In the fully pipelined architecture, the behavior of each folded butterfly execution unit can be represented as a C function that takes one sample as input and produces one sample as output, using a twiddle table, a memory and some control information represented as global variables. This reusable C-function can easily represent all the stages. Then for fully pipelined hardware implementation, the functional description of each stage can be wrapped inside a SystemC bit level interface and synthesized using any SystemC based HLS tool. As discussed before, the sequential C representation with double nested loops allows synthesis of resource shared architectures with less area, while this stage-level model allows synthesis of a pipelined architecture with higher throughput. Apparently it may look like that these two approaches have opposite limitations, one being suited for very low throughput and the other for very high throughput. But the second description, in which the FFT is partitioned and described as C-function calls with partitioned memory, can be useful to obtain variable concurrency FFT implementations starting from the same representation.

As mentioned above, HLS tools cannot easily increase the level of concurrency automatically, but they can easily remove it, by inlining functions and scheduling them on the same resources. They also allow one to merge different memories into a single one. Using these two mechanisms, the functional description of the pipelined FFT can be mapped to different micro and macro architectures. The macro architecture can be changed e.g. by changing the level of concurrency, that is by mapping different butterfly stages to a single SystemC thread. The micro-architecture can be changed by modifying the HLS constraints that describe the set of available resources and how the butterfly computation will be scheduled on them. In other words, the functional C description of the FFT can be mapped to a fully sequential architecture by

— wrapping it inside a single SystemC thread,
— merging and mapping all the C arrays into a single memory.

In Figure-13 each FFT stage has a buffer with a size of $\frac{N}{2^m}$ samples, where N is the FFT length and m is the stage number starting from the left. The total memory required for storage is $(N-1)$ samples, which is the sum of the memory sizes required by all the butterfly stages. For a pipelined implementation with the highest level of concurrency, each buffer should be mapped to a separate SRAM with one read and one write port. For lower levels of concurrency, the FFT stages should be merged into groups of two, four and so on, where each group is then mapped to a single thread. The FFT is initially represented at the functional level in a partitioned manner, by mapping each butterfly stage to a separate C function with a buffer for storage. So during the process of merging FFT butterfly stages in groups, the memories in each group are also merged to save chip area. Figure-14 illustrates how the memories are merged into groups for different levels of concurrency.

7. HLS AND DESIGN EXPLORATION USING FFT HLS-IP

We have taken an FFT of length 256 with a 16-bit data path and synthesized it using UMC-90nm libraries for different levels of concurrency, ranging from fully sequential

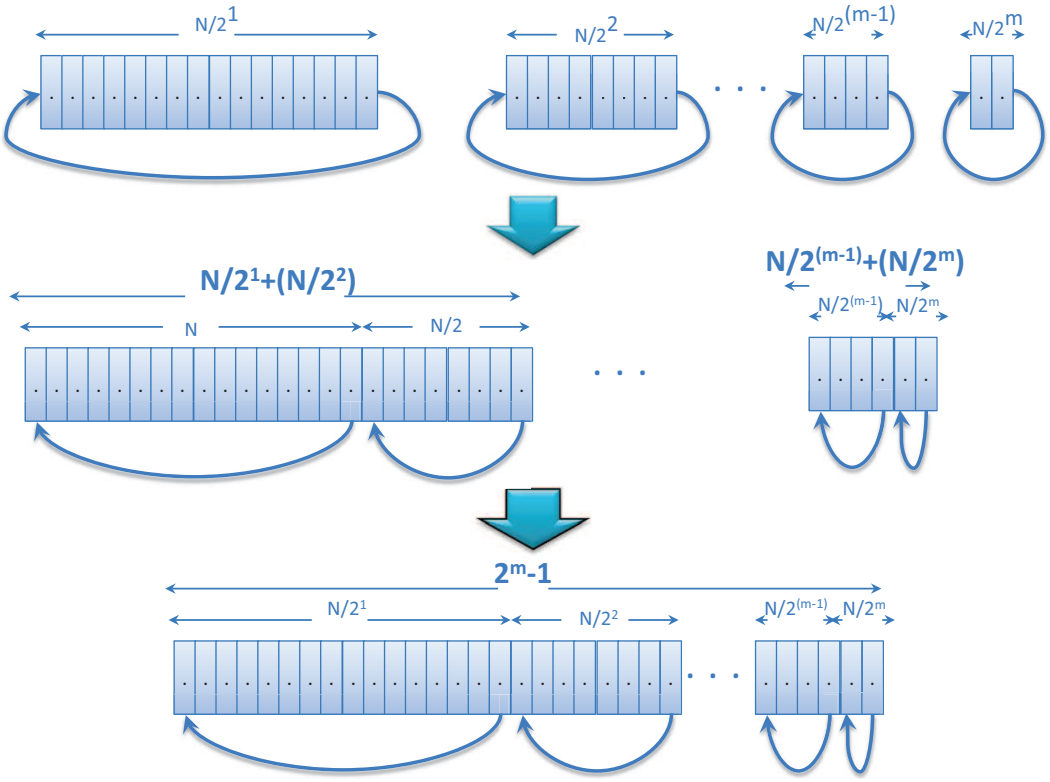

Fig. 14. Merging of memories for different levels of concurrency

to fully pipelined implementations. The HLS-IP C description was generated using our HLS-IP generator. Then this model was used for HLS by using various tools, namely Cadence C-to-Silicon Compiler and Calypto Catapult, as well as Synopsys Synphony and Mathworks HDL Coder. The HLS-IP Generator can also generate SystemC wrappers for the different concurrency levels, as mentioned above, by stitching different function calls (which correspond to different folded stages) into a single thread.

The code listed in Algorithm-3 shows a thread wrapper with a concurrency level of 4 for an FFT of length 256. The 256 point FFT has 8 stages in total, so 2 stages are grouped into a single thread. Similarly the listing in Algorithm-2 shows a thread wrapper when all butterfly stages are concurrently executing, with each stage wrapped inside a separate thread.

Figure-15 shows how throughput scales when the concurrency level is changed from 1 to 8. The concurrency level of 1 (a fully resource shared architecture) is synthesized by merging all the memories to a single memory and by wrapping all the butterfly stages into a single thread, whereas the concurrency level of 8 corresponds to a pipelined architecture with each butterfly stage wrapped inside a separate SystemC thread. The design was synthesized at 200MHz, obtaining a minimum throughput of 25 Million Samples Per Second (MSPS) with concurrency 1 and a maximum throughput of 200 MSPS (i.e. one sample per clock cycle) with concurrency 8. Figure-16 shows the area required for different throughput levels.

Such a large variation in throughput and required hardware resources for implementation as shown in Figure-15 and Figure-16 cannot be obtained from any sequen-

ALGORITHM 2: SystemC Thread Wrapper for concurrency level-8, for 256 point FFT

```
void pipeLinedFFT_WRAPPER::stageN_exec_WRAPPER(void)
{
        struct mcomplex input_sample;
        struct mcomplex output_calculated;
        hshake[STAGE_NUMBER_sN].write(false);
        wait();
        init_stageN();
        wait();
        while(hshake[STAGE_NUMBER_sN-1].read()==false) wait();
        hshake[STAGE_NUMBER_sN].write(true);
        while(1)
        {
                input_sample.real=(int)in_interfaceN_real.read();
                input_sample.imag=(int)in_interfaceN_imag.read();
                functional_behavior_stageN(input_sample,&
                    output_calculated);
                out_interface_real[STAGE_NUMBER_sN].write(
                    output_calculated.real);
                out_interface_imag[STAGE_NUMBER_sN].write(
                    output_calculated.imag);
        }
}
```

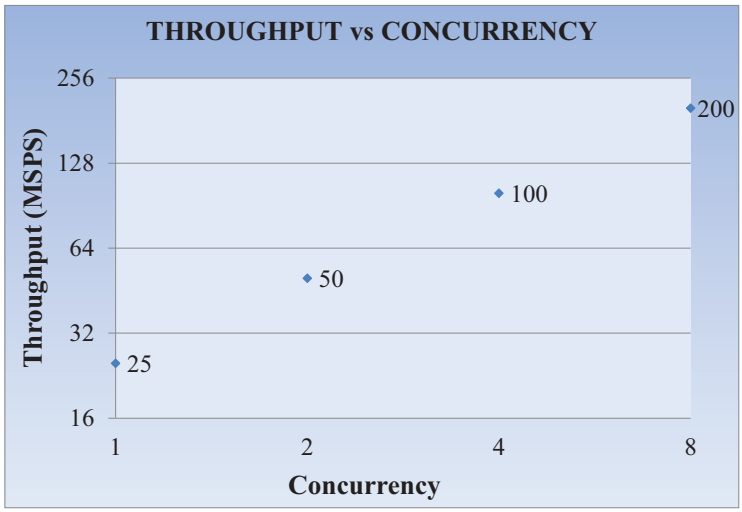

Fig. 15. FFT Throughput vs. Concurrency

tial description of the FFT in C. It is only made possible by the memory and code partitioning strategy that we used to interleave the butterfly stages.

The results obtained from our HLS-IP generator were also compared with similar implementations obtained from other tools. Synphony Model Compiler is a tool from Synopsys that can generate RTL code from Simulink models. It has a library of components that can be used in Simulink for verification, but it supports generation of RTL only for blocks that come from its own IP library. Synphony can only generate a fully

ALGORITHM 3: SystemC Thread Wrapper for concurrency level-4, for 256 point FFT

```
void FFT_WRAPPER::stageN_M_exec_WRAPPER(void)
{
        struct mcomplex in,outN,outM;
        struct mcomplex ind,outNd,outMd;
        sc_int < 16 > trans_real;
        sc_int < 16 > trans_imag;
        init_stageN&M();
        wait();
        while(start.read()==false)wait();
        while(1)
        {
                trans_real=(int)in_interfaceNM_real.read();
                trans_imag=(int)in_interfaceNM_imag.read();
                in.real=trans_real;
                in.imag=trans_imag;
                functional_behavior_stageN(in,&outNd);
                trans_real=outNd.real;
                trans_imag=outNd.imag;
                outN.real=trans_real;
                outN.imag=trans_imag;
                wait();
                functional_behavior_stageM(outN,&outMd);
                trans_real=outMd.real;
                trans_imag=outMd.imag;
                outM.real = trans_real;
                outM.imag=trans_imag;
                out_interfaceNM_real.write(outM.real);
                out_interfaceNM_imag.write(outM.imag);
                wait();
        }
}
```

pipelined FFT that is very close, in terms of both performance and area, to the fully concurrent implementation obtained from our HLS-IP generator.

Similarly, we generated an FFT implementation from HDL Coder . It generates a sequential resource shared architecture, which in our case corresponds to concurrency level-1. Figure-16 compares the different implementations obtained from our HLS-IP Generator with those obtained from Synphony Model Compiler and HDL Coder. From these results it is clear that the Quality of Results (QoR) of our FFT implementations is very close to that of these other tools, which rely on hand-optimized parameterized architectures, while our solution allows a much broader range of implementations from a single model.

8. USE CASES

Different use-cases have been selected from different digital signal processing application domains and FFT HLS-IP is generated and optimally synthesized to meet different cost and performance requirements. The quality of result (QoR) for our automatically generated FFT HLS-IPs is compared with manually designed FFT IPs and also with the IPs generated from Simulink using the HDL Coder tool.

To demonstrate the effectiveness of our design methodology for HLS-IP blocks to be used in a model-based hardware design environment, we considered very different

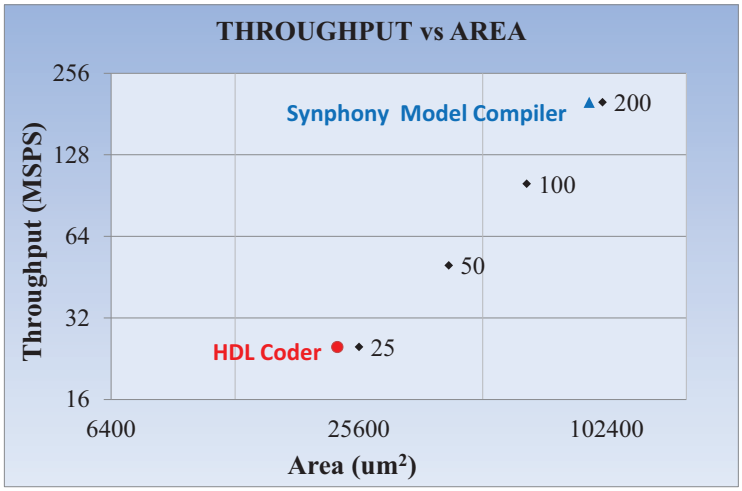

Fig. 16. FFT Throughput vs. Chip Area

use cases for our FFT HLS-IP. They include an FFT processor for frequency domain audio detection,a GPS acquisition front end, a DSP front-end for a radar and an OFDM receiver. The use-cases have been synthesized for two different target implementation technologies, namely the 90nm ASIC technology libraries from UMC and the Kintex-7 FPGA from Xilinx

8.1. Frequency Domain Audio Detector

In sound-triggered wireless security camera applications, a front-end audio detector is employed at the start of the alarm processing chain. Still images and video streams are very expensive to collect and process, hence the video cameras are only turned on when the low-power sound sub-system detects an event of interest, as illustrated in Figure-17.

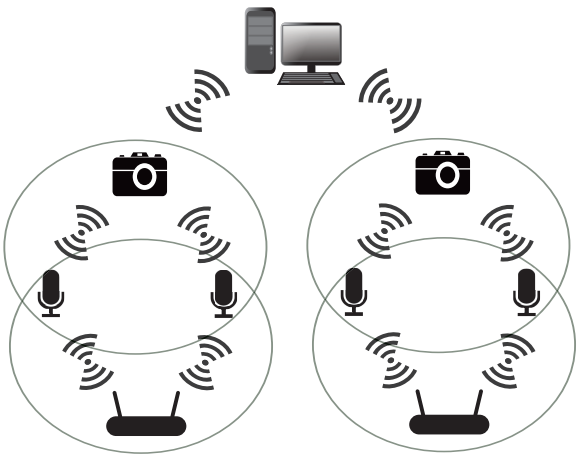

Fig. 17. Sound triggered multi-modal wireless surveillance network

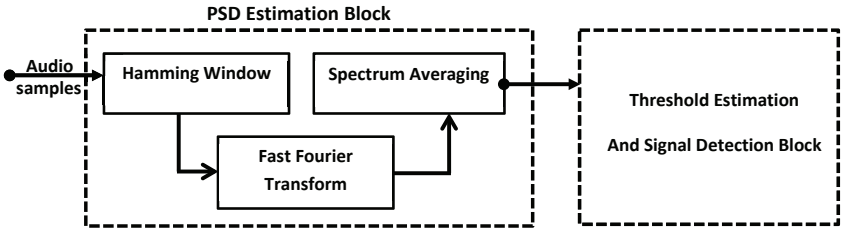

Fig. 18. Front end audio detection algorithm block diagram

A frequency domain audio detector is considered in this case, as shown in Figure-18. It consists of two main blocks: "power spectral density estimation" (PSD) and "threshold estimation and signal detection" that we fully modeled in Simulink.

After code generation from Simulink and profiling, the FFT within PSD estimation is identified as the power and performance bottleneck, and thus selected for HW implementation. We then compare the results of HLS when performed on our IP and when performed on the C code automatically generated by RTW.

In this use case, due to the relatively low frequency of the audio input, performance requirements are low, namely one 256-sample FFT with 14 bits of precision every 4 milli-seconds. Hence the goal is essentially area and power optimization.

Table-I compares the implementation results obtained after HLS and RTL level power estimation of our IP block and of the default RTW C code implementation, also encapsulated in SystemC. Our (hardware-oriented) IP consumes 45 percent less area and has essentially the same throughput as the (software-oriented) RTW code. The power consumption is also lower in our case.

Table I. RTW FFT vs. FFT HLS-IP for audio detector application

Design	Area (mm^2)	Power (mW)
RTW FFT	0.60	4.14
FFT HLS-IP	0.42	2.16

Table II. Comparison of implementation cost for different audio detector blocks Implemented using HLS

Audio Detector Blocks	Area (mm^2)
FFT HLS-IP	0.42
Windowing and Spectrum Averaging	0.1783
Threshold Estimation and Detection	0.267

One important reason for the area difference is the better bit-width optimizations that are enabled by our bit-accurate representation of the individual butterflies. Both the data path and the memories were trimmed to the exact 14 bit width required by the usage scenario, instead of the default 16 bits used by the RTW implementation (which is limited to C data types).

The complete audio detection hardware was implemented using high level synthesis, using the RTW code for the other blocks. Table-II compares the area cost of the FFT implementation to the other blocks. Their relatively lower area implies that the development of a dedicated HLS-IP for them may not be justified in terms of manpower expenditure

8.2. FFT-based GPS Acquisition

A GPS receiver must be able to capture and demodulate signals transmitted by at least four GPS satellites. Every satellite convolves its signal with a code that spreads its power over a relatively large spectrum. When a GPS receiver is turned on, it first task is to identify which satellites are visible at that time and place. This means sampling the received signal and figuring out which codes have been used by the visible satellites for spreading and what is the approximate phase of the (known) spreading sequence at that time. Because of the relative movement of satellites with respect to the receiver, there is also a Doppler frequency shift which the receiver is required to estimate.

There are many different techniques for acquisition, including parallel search techniques based on FFT. The FFT-based GPS acquisition algorithm block diagram is shown in Figure-19. The first block in the chain performs two tasks. First it down converts data to baseband, and then it averages data samples to have a lower sampling rate at the output. The next block performs the frequency domain transformation using the FFT algorithm. Then the data is multiplied by the transform of the spreading code. This is followed by an inverse FFT and the peak search in the time domain.

We modeled this complete algorithm in the Simulink environment, verifying it with both the native Simulink FFT block and the S-function FFT HLS-IP developed by us. Then we synthesized both FFTs (our HLS-IP and the RTW version) under the very stringent timing constraints required by this usage scenario, namely one 1024-sample FFT with 4 bits of precision every milli-second.

Note that the throughput is 16 times higher than in the frequency domain audio detector case, but the data-path requires only about 1/4 of the bits of the previous case (4 versus 14).

Table III. RTW FFT vs. FFT HLS-IP for GPS acquisition application

Design	Area (mm^2)	Power (mW)
RTW FFT	0.87	6.8
FFT HLS-IP	0.60	5.7

Table-III compares the results of these two implementations. Our IP is much better in terms of area than the RTW version, with essentially the same throughput. Power consumption is also smaller in our case.

We also compare our implementation with a manually optimized RTL that was specifically designed for FPGA implementation in [Molino et al. 2008], while our IP

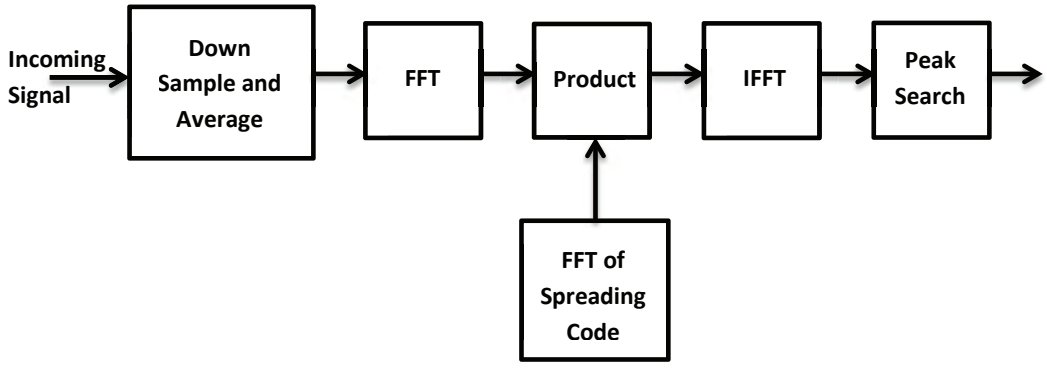

Fig. 19. Core GPS acquisition algorithm

model was not specifically tuned for FPGAs but only for generic HW implementation. The results are reported in Table-IV, showing that our implementation is comparable with a hand optimized RTL. The SRAM requirements are exactly the same, while we are about 20% worse in terms of area. For the GPS receiver use-case, we present results only for the FFT block, because it is used multiple times during the execution of algorithm and it is by far the most complex and resource consuming block.

Table IV. Hand optimized RTL implementation vs. FFT HLS-IP for GPS acquisition application

Design	Number of Slices	SRAM ($Kbits$)
Manual Design	$4.2K$	18
FFT HLS-IP	$5.2K$	18

8.3. FFT-based Digital Signal Processing Unit for Radar

A radar is an electronic device that is used for estimating different parameters (e.g. speed, direction and position) related to the movement of an object. Radars typically find uses in military and commercial applications. In particular, they are an essential component of assisted driving applications in automotive electronics, e.g. parking assistance, lane departure warning and collision avoidance. In this case study we experimented with the Digital Signal Processing (DSP) unit of a radar for automotive applications based on the Continuous Wave Frequency Modulation (CWFM) technique. The CWFM based radar transmits a frequency-modulated signal that is reflected from a target object. The reflected signal is captured and different parameters, such as the time of flight and Doppler shift are estimated, as shown in Figure-20. Then these can be translated into the distance and the velocity of the object.

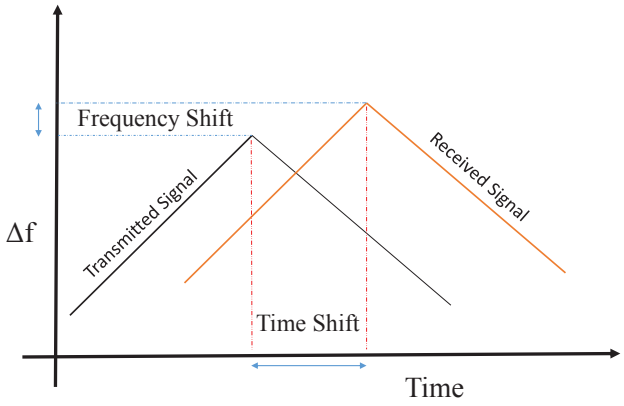

Fig. 20. Operation of CWFM Doppler Radar

In our experiments we modeled using Simulink the digital signal processing unit of this radar, as shown in Figure-21. The most expensive block, also in this case, is a high precision 2048 sample FFT. In this experiment we targeted the implementation to a Kintex-7 FPGA (xc7k160tfbg484-3) from Xilinx. Table-V shows the synthesis results only for the FFT, using the performance requirement of the full radar application. It illustrates that our HLS-IP, synthesized using Calypto Catapult, uses similar resources when compared to the optimized RTL implementation from HDL coder, for the same real-time throughput constraints. Note that the LUT cost obtained via HLS uses more

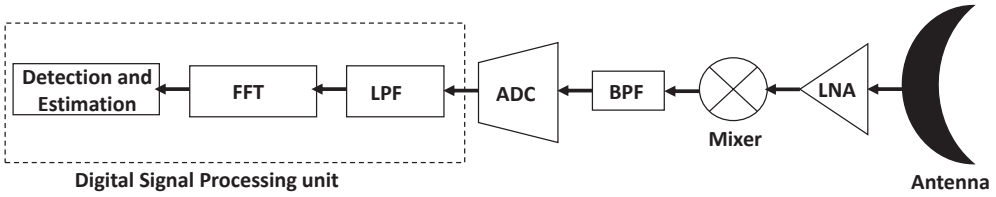

Fig. 21. CWFM Radar Receiver Architecture

Table V. FFT Implementation for Radar DSP Unit HLS-IP vs. HDL Coder

Synthesis Tool	DSP48	LUT	FF	Memory Blocks
Catapult (HLS-IP)	80	9069	7015	22
HDL Coder (Manually Designed RTL IP)	72	7744	11524	25

LUTs because the resulting RTL is less efficient for exploiting the DSP48 units. This is something that we will consider for the future, e.g. by automatically generating FPGA-specific mapping directives for the HLS tool.

Table VI. Comparison of implementation cost for different radar blocks Implemented using HLS

Radar Block	DSP48	LUT	FF	Memory Blocks
FFT	80	9069	7015	22
LPF	6	901	1529	0
Detection & Estimation	24	11858	4927	0

Table VII. Full Radar DSP Unit Implementation HLS vs. HDL Coder

Synthesis Tool	DSP48	LUT	FF	Memory Blocks
Catapult/HLS	110	21828	13471	22
HDL Coder	288	13268	11878	25

The complete synthesis result for radar DSP front-end are reported in Table-VI. As shown in Table-VII, the results obtained using HLS-IP and high level synthesis are very comparable with the results obtained using HDL Coder. The HDL Coder based solution uses many more DSP48 slices the result of HLS-IP,which again uses many more LUTs. The other blocks are synthesized starting from the automatically generated C-code produced by Embedded Real Time Coder.

8.4. FFT-based OFDM Receiver

Orthogonal frequency division multiplexing (OFDM) is an extensively used digital encoding technique used for baseband modulation in wireless networks. Several IEEE standard, like 802.11a and 802.11b, use OFDM. Figure-22 shows a simplified block diagram (blocks for channel coding part are omitted and only core modulation blocks are shown) of an OFDM receiver. All the other blocks are much simpler than the FFT from the hardware design point of view, and again can be synthesized from the ERT-generated code. The FFT HLS-IP was parameterized and synthesized considering a 9Mbps data rate specification. The authors of [Shao and Slump 2008] suggested that 5-bits are sufficient for sampling the OFDM receiver input in baseband. So the datapath for FFT is parameterized for an input sample bit-width of 5. Scaling is used and the integer part of the output results is allowed to grow by one bit after each stage of butterflies, resulting in a final output with 11 bits.

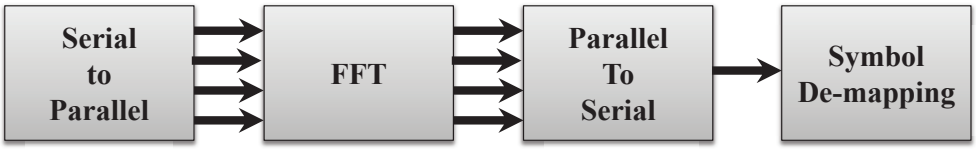

Fig. 22. OFDM receiver simplified block diagram

Table VIII. FFT HLS-IP Synthesis results for an OFDM receiver

Implementation Type	Memory LUT	DSP48	LUT	FF
HLS-IP	0	0	1901	1399
Simulink HDL Coder	150	0	1558	2122

Table-VIII gives implementation results for the FFT processor synthesized from Simulink HDL Coder and our HLS-IP. The FFT implementation obtained from our HLS-IP uses almost 52 % less flip-flips but uses 18% more LUTs. The reason for this increased use of LUTs is that for HLS-IP the read only storage is synthesized from LUTs, wheres in the case of HDL coder it consumes 150 memory LUTs.

9. CONCLUSIONS

In this paper we demonstrated how one can design a parameterized bit-true IP, coded in plain C, which can be both integrated with Simulink for verification, and used for High-Level Synthesis into hardware. To prove the effectiveness of our approach we also compared our synthesis results with other implementations that are based on hand optimized RTL, parameterized RTL, and generating C code automatically from Simulink models. We also illustrated our HLS-IP design flow by using a concrete example, namely a flexible HLS-IP model of a Fast Fourier Transform. It allows one to model the FFT as a symmetrically partitioned C-model that can be used for simulation in the Simulink environment, and then for efficient high level hardware synthesis. We can synthesize FFT hardware with different optimized datapath widths and with a great degree of throughput variations which is achieved by changing the concurrency of the hardware implementation, both in our generator (for thread-level parallelism) and in the HLS tool (for loop-level parallelism). We implemented an FFT HLS-IP Generator that takes as input a configuration file specifying the required FFT length and the number of threads to be used. This FFT HLS-IP is used to show the effectiveness of our design flow. It generates a functional level C description of the FFT, as well as the synthesis scripts, the S-function wrapper for the Simulink environment and the SystemC wrapper for high level hardware synthesis. Finally we generated and synthesized our FFT HLS-IP with different architectures for different performance and resource requirements, for very different applications starting from the same functional C template, and we compared the quality of results with other tools.

References

Fft logicore - xilinx fft ip generator.

Fft megacore function - fft megacore® function:a high-performance, highly parameterizable fft processor.

Labview system design software.

Real-time workshop: Generates c/c++ from simulink models.

Simulink - simulink is a block diagram environment for multidomain simulation and model-based design.

Simulink hdl coder - generate hdl code from simulink models and matlab code.

Synphony model compiler.

BUTT, S. AND LAVAGNO, L. 2012a. Design space exploration and synthesis for digital signal processing algorithms from simulink models. In *Design and Test Symposium (IDT)*.

BUTT, S., SAYYAH, P., AND LAVAGNO, L. 2011. Model-based hardware/software synthesis for wireless sensor network applications. In *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*. 1–6.

BUTT, S. A. AND LAVAGNO, L. 2012b. Designing parameterized signal processing ips for high level synthesis in a model based design environment. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. CODES+ISSS '12. ACM, New York, NY, USA, 295–304.

BUTT, S. A., MANCINI, S., ROUSSEAU, F., AND LAVAGNO, L. 2014. Design of a pseudo-log image transform hardware accelerator in a high-level synthesis-based memory management framework. *Journal of Electronic Imaging 23,* 5, 053012–053012.

CHOULIARAS, V., GALIATSATOS, P., NAKOS, K., REISIS, D., AND VLASSOPOULOS, N. 2009. Efficient cascaded vlsi fft architecture for ofdm systems. In *Electronics, Circuits, and Systems, 2009. ICECS 2009. 16th IEEE International Conference on*. 97 –100.

DAVE, N., PELLAUER, M., GERDING, S., AND ARVIND. 2006. 802.11a transmitter: A case study in microarchitectural exploration. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06. Proceedings.* MEMOCODE '06. IEEE Computer Society, Washington, DC, USA, 59–68.

HAUBELT, C., SCHLICHTER, T., KEINERT, J., AND MEREDITH, M. 2008. Systemcodesigner: Automatic design space exploration and rapid prototyping from behavioral models. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. 580–585.

HUANG, K., IL HAN, S., POPOVICI, K., BRISOLARA, L., GUERIN, X., LI, L., YAN, X., CHAE, S.-I., CARRO, L., AND JERRAYA, A. 2007. Simulink-based mpsoc design flow: Case study of motion-jpeg and h.264. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*. 39–42.

KEE, H., PETERSEN, N., KORNERUP, J., AND BHATTACHARYYA, S. 2008. Systematic generation of fpga-based fft implementations. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*. 1413 –1416.

KIENHUIS, B., RIJPKEMA, E., AND DEPRETTERE, E. 2000. Compaan: deriving process networks from matlab for embedded signal processing architectures. In *Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on*. 13–17.

LEE, S.-Y. AND CHEN, C.-C. 2006a. Vlsi implementation of programmable fft architectures for ofdm communication system. In *Proceedings of the 2006 international conference on Wireless communications and mobile computing*. IWCMC '06. ACM, New York, NY, USA, 893–898.

LEE, S.-Y. AND CHEN, C.-C. 2006b. Vlsi implementation of programmable fft architectures for ofdm communication system. In *Proceedings of the 2006 international conference on Wireless communications and mobile computing*. IWCMC '06. ACM, New York, NY, USA, 893–898.

MITRA, S. 1999. Xcc-a tool for designing parameterizable ip cores in vhdl. In *Signals, Systems, and Computers, 1999. Conference Record of the Thirty-Third Asilomar Conference on*. Vol. 1. 752 –756 vol.1.

MOLINO, A., GIRAU, G., NICOLA, M., FANTINO, M., AND PINI, M. 2008. Evaluation of a fft-based acquisition in real time hardware and software gnss receivers. In *Spread Spectrum Techniques and Applications, 2008. ISSSTA '08. IEEE 10th International Symposium on*. 32–36.

MURPHY, G., POPOVICI, E., BRESNAN, R., MARNANE, W., AND FITZPATRICK, P. 2004. Design and implementation of a parameterizable ldpc decoder ip core. In *Microelectronics, 2004. 24th International Conference on*. Vol. 2. 747 – 750 vol.2.

NIKOLIC, G. April 2011. *Fourier Transforms - Approach to Scientific Principles*. Intech Open.

SAYYAH, P., BUTT, S., AND LAVAGNO, L. 2011. Simulink-based hardware/software trade-off analysis technique. In *Applied Electrical Engineering and Computing Technologies (AEECT), 2011 IEEE Jordan Conference on*. 1 –7.

SHAO, X. AND SLUMP, C. H. 2008. Quantization effects in ofdm systems.

TAKACH, A. 2010. Creating c++ ip for high performance hardware implementation of ffts. In *DesignCon*.

TOLEDO, A., SUARDIAZ, J., CUENCA, S., AND GREDIAGA, A. 2006. Novel simulink blockset for image processing codesign. In *Electrotechnical Conference, 2006. MELECON 2006. IEEE Mediterranean*. 117–120.

WERNSING, J. R. AND STITT, G. 2010. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. *SIGPLAN Not. 45,* 4, 115–124.

Received May 2015; revised May 2015; accepted May 2015